Automated sensor data extraction pipeline and Subsystems visualization for EDEN ISS Project

by

Rohit Paudel

Bachelor Thesis in Computer Science

in collaboration with,

Deutsches Zentrum für Luft- und Raumfahrt - EDEN ISS Project

Prof.Franscesco. Maurelli
Bachelor Thesis Supervisor
Jacobs Universiy Bremen

Conrad Zadler
Bachelor Thesis Supervisor
DLR Bremen

Date of Submission: August 31, 2020

Jacobs University — Computer Science and Electrical Engineering

With my signature, I certify that this thesis has been written by me using only the indicates resources and materials. Where I have presented data and results, the data and results are complete, genuine, and have been obtained by me unless otherwise acknowledged; where my results derive from computer programs, these computer programs have been written by me unless otherwise acknowledged. I further confirm that this thesis has not been submitted, either in part or as a whole, for any other academic degree at this or another institution.

Signature

Bremen, 30-08-2020

Place, Date

Abstract

The EDEN ISS project has deployed a Mobile Test Facility in Antarctica and working to develop plant cultivation technologies suitable for Space and future interplanetary travel missions. The EDEN ISS project demonstrates the operational capability of key technologies and collects relevant data. The collected data can be used to further develop new cutting edge technologies in this field to improve the rate of innovation. As per the problems faced during the winter-over 2018 operation season, a number of problems were faced in Mobile Test Facility. A need for a software tool to monitor the subsystems and visualize the relation and interaction with the used sensor component was felt. This paper explores to design and implement a software tool, with the guidelines and requirements provided, to enable better monitoring of the Mobile Test Facility in Antarctica from the EDEN ISS mission control room and project partners all around the world.

Contents

1 Introduction 1

2 EDEN ISS 1
- 2.1 MTF Structure . . . 2
- 2.2 Subsystems . . . 3
  - 2.2.1 Power Control and Distribution System . . . 3
  - 2.2.2 Atmosphere Management System . . . 3
  - 2.2.3 Nutrient Delivery System . . . 3
  - 2.2.4 Thermal Control System . . . 4
  - 2.2.5 Command and Data handling System . . . 4

3 Use case and Guidelines 4

4 Tools and Technologies 5
- 4.1 Laravel Web Framework . . . 6
- 4.2 Guzzle . . . 7
- 4.3 React . . . 7

5 Design and Architecture 7

6 Implementation 8
- 6.1 Understanding ARGUS API . . . 8
- 6.2 Backend System . . . 10
  - 6.2.1 Database Migration . . . 10
  - 6.2.2 Eloquent ORM (Model) and relationship . . . 12
  - 6.2.3 API Call service . . . 13
  - 6.2.4 JSON Parser Service . . . 14
  - 6.2.5 Storing Parsed Data . . . 16
  - 6.2.6 Periodic APIs call . . . 18
  - 6.2.7 RESTful API Infrastructure . . . 19
- 6.3 Frontend System . . . 21
  - 6.3.1 Calling RESTful API . . . 22
  - 6.3.2 Layout Visualization . . . 22

7 Testing 23

8 Conclusions 24

9 Future Work 24

1 Introduction

The main goal of this research project is to explore and implement automated sensor data extraction pipeline and control system visualization strategy for EDEN ISS Project. The extracted data need to be stored locally in DLR, Bremen. Then, the data need to be visualized within the control system workflow layouts using a web based GUI. This paper discuss the details on how experiment with different technologies are carried out to achieve the state of the art software infrastructure and meet the research project goal.

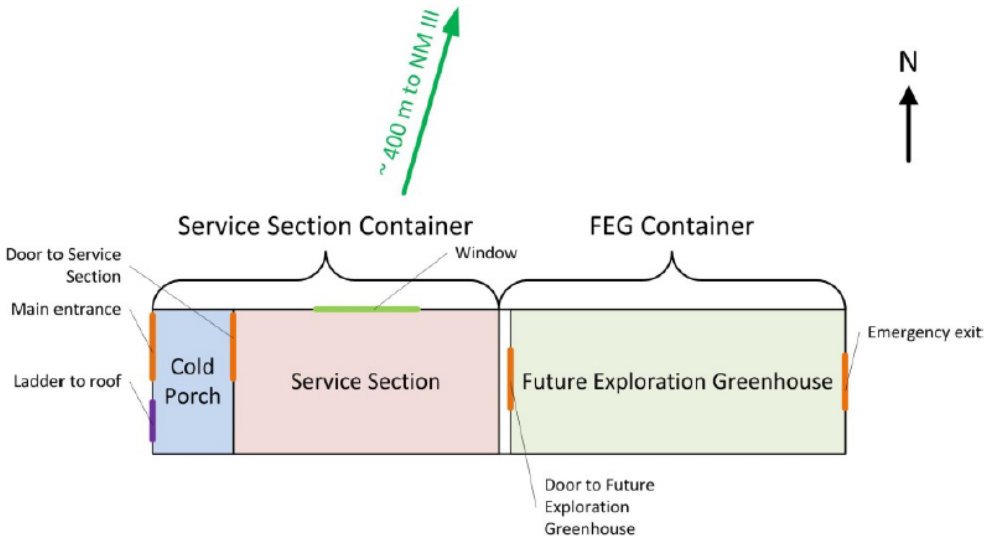
Plant cultivation in large-scale closed environments is challenging and several key technologies necessary for space-based plant production are not yet space-qualified or remain in early stages of development [17]. EDEN ISS project was started in March 2015 with the purpose to further develop plant cultivation technologies and operations in space [17]. The leading organization DLR along with fourteen partners from eight countries in Europe, Canada and the US joined forces to build a space analogue Mobile Test Facility (MTF) and was successfuly deployed 400 meters south from German Neumayer III Antarctic research station in January 2018. The greenhouse was operated for 9 months continuously until mid of November the same year producing 270 kg of fresh edible biomass linked with more than 20 different crops [17] [16]. The Neumayer III analogue test site provides the opportunity to validate complex integrated systems outside the laboratory[16]. The analogue deployment serves to confirm the functionality of concepts and technologies, adding another degree of security and risk mitigation for future ISS and interplanetary surface bio-regenerative life support systems (BLSS).

The Future Exploration Greenhouse (FEG) among three sections of MTF, is the main plant growth area of the MTF, consisting of multilevel plant growth racks operating in a precisely controlled environment[16]. Plants are grown based on existing hydroponic concept, developed by DLR Institute of Space Systems, that is a hybridization of classic NFT hydroponics and aeroponics[15]. There are altogether five subsystems in MTF that is, power control and distribution system (PCDS), atmosphere management system (AMS), nutrient delivery system (NDS), thermal control system (TCS) and command and data handling system (CDHS). The different components of each sub system are provided and/or developed in collaboration with the partners. There are altogether 121 sensors and 99 actuators incorporated in the MTF within all the subsystems. All the sensors and actuators are connected to CDHS's Argus system, which logs the collected data and support in maintaining and monitoring of MTF environment from the site and remotely. There is wireless connection between MTF and Neumayer III. The Neumayer III communicate with satellite communication technology, giving ability to access data from ground station in Europe.

ARGUS systems provides API (Application Program Interface) to access the sensor and actuators parameters and data remotely. This API is used internally by ARGUS software for visualization and monitoring software. The ARGUS System API is the data source for my data extraction pipeline.

2 EDEN ISS

This section explain the parts of EDEN ISS Project which will lay down the foundation to understand the use case for the software project covered in this thesis project.



[15]

Figure 1: Overview of the EDEN ISS MTF main elements.

The EDEN ISS MTF consists of two customized 20 foot high-cube shipping containers. The two container approach was necessary because of the logistics requirement of MTF from Germany to the Neumayer III research station in Antarctica[15].

2.1 MTF Structure

The MTF is divided into three distinct sections as shown in the figure above. They are explained in brief below.

- Cold porch: It is a small room providing storage and acting as a buffer to prevent the entry of cold air into the plant cultivation and main working areas when the main entrance door of the facility is utilized[16]. A fresh water and a waste water tank, both capable of holding up to 250 L, are located underneath a raised floor system. The fresh air supply system is also located in this section. They are part of nutrition delivery subsystem.
- Service section: This section have the primary system control, air management, thermal control, and nutrition delivery system of the MTF together with the international standard payload rack (ISPR) plant growth demonstrator[16]. The subsystems within the Service Section ensure that the approximately 12.5 meter squares of cultivation area in the second container, the Future Exploration Greenhouse, have the proper environmental conditions, nutrients and illumination for optimal crop growth[15].
- Future Exploration Greenhouse (FEG): This is the main plant growth area of the MTF, a center aisle fixed shelves design (corridor centered, shelves on each side), consisting of multilevel plant growth racks operating in a precisely controlled environment[16]. There are four racks per side separated by 1000 mm wide corridor. The FEG has racks with one, two and four levels to allow the volume optimized cultivation of different plant species. The FEG is considered a closed system.

2.2 Subsystems

2.2.1 Power Control and Distribution System

The required power for MTF is received from Neumayer III which will be distributed among different subsystems and appliances by power control and distribution system (PCDS). The average power demand of the MTF calculated over one year is around 11.5kW. However, the plants require a certain photoperiod-darkperiod cycle which results in a different power demand during photoperiod and darkperiod. Uninterruptible power supply (UPS), is installed within the MTF service section to handle potential power shortage and voltage fluctuations to avoid uncontrolled shutdown of the command and data handling system and communication between the MTF and Neumayer[15].

2.2.2 Atmosphere Management System

Numerous sensors and actuators of different kinds are integrated in each subsystems to create feedback loop providing favorable conditions for plants to grow inside FEG. Since only one operator will handle the operation of the FEG, there is a need for objective assessment of plant welfare and performance, through automatic detection and remote expert assessment of the information[17].The atmosphere management system provides fresh air, relatively cold, dry and rich in CO_2. The gas exchanges between the environment and the crops, as well as the thermal loads which acts on the air, result in a deviation of the climatic conditions from the desired set points. Respective sensors are used to measure relative humidity, temperature, CO_2 levels, O_2 levels and filtered and dehumidified air is circulated to correct the parameter fluctuations back to desired set points.

The plant cultivation racks inside the FEG are outfitted with water-cooled LED lamps, altogether 42 lamps, and two per each level. Each lamp has LEDs in four different wavelenghts (red: 630 nm, blue: 450nm, far-red: 735nm, white 5700 K) which can be controlled independently in 0.1 % increments from 0-100% output. Furthermore, the illumination schedules can be individually programmed for each lamp. The LEDs also include a thermal runaway cut-off switch, which will shut the LED off should its temperature rise above 50℃ [17].

2.2.3 Nutrient Delivery System

The nutrient delivery system consist of two separate solution loops, each with its own set of feed and drainage pipes in the subfloor of the FEG. Three way valves in the subfloor are used to define which nutrient solution is fed to different racks. One high-pressure pump (HPP) is able to supply all plant cultivation trays in one rack. The setup with two tanks and their own distribution piping provides the possibility to have two seperate nutrient solutions with different nutrient compositions, EC and pH value. In this subsystem, sensors like pH sensor, electric conductivity (EC) sensor, pressure sensor, valves, flow meter, level sensor have been used. Dosing pump is used to maintain the pH of the solution, i.e. by controlling acid or base addition rate in the solution[17]. The eitire NDS solution loop is closed (recirculating)[15].

2.2.4 Thermal Control System

The thermal control system has to actively dissipate the bulk of the head produced inside the entire MTF. Specially, the thermal system will dissipate the heat form three sources: the ISPR, the AMS and the LED panels in the FEG[17]. The Argus Titan Omni Sensor aspirated boxes consist of temperature, relative humidity and CO_2 concentration sensor. In addition to two boxes, four EE071 temperature and relative humidity sensors are placed at various locations in the FEG. The data from these sensors is the source for a feedback loop to control the temperature and other parameters related to AMS and TCS inside FEG.

2.2.5 Command and Data handling System

The most relevant among all the subsystem in context of this paper is command and data handling system. CDHS is based on Argus system provided by Argus Controls(Surrey, BC, Canada) which includes a complete hardware and software solution for monitoring and equipment automation purposes[15]. The total number of 121 sensors and 99 actuators are connected to ARGUS system. The ARGUS Server PC, including RAID (Redundant Array of Independent Disks) system, is used to control and monitor all the systems inside and outside the MTF except for the ISPR, the safety system and the camera system. There are 36 cameras controlled by dedicated Camera Control PC. There is wireless connection between the MTF and the NM III enabling data transfer. The patch antenaa is installed on the roof of the facility which communicates through line of sight to a matching antenna installed on NM III[15]. The data from all the connected sensors and actuators can be accessed using Virtual Network of Alfred-Wegener-Institute for Polar and Marine Research (AWI) which operate the Neumayer III station. During nominal operation of the MTF and the NM III, a data rate of 100 kbps will be available to the EDEN ISS project for data transfer to the ground station in Europe. For off-nominal situations, such as video conferencing, a maximum data rate of 300 kbps will be provided.

Once connected to the Virtual Network of Alfred-Wegener-Institute, the ARGUS API(Application Program Interface) can be called. ARGUS System has a SQL database with all the sensors parameters and data. ARGUS API allows access to the data from external sources.

3 Use case and Guidelines

This section gives an overview of the failures of EDEN ISS FEG in last deployment, problems faced, relevance and requirements of the software project that will be designed and implemented as the part of thesis project.

The EDEN ISS project demonstrate operational capability of key technologies in an environment, similar in certain relevant characteristics to space. A remote monitoring and commanding system is a necessity for the project. A number of technical system failures happened in FEG during the winter-over 2018 season. The Thermal control system caused most of the technical problems which is responsible to provide cooling fluid to the LED lamps, the ISPR-like cultivation system, and the humidity recover system. Most of the problems were associated with a poor subsystem design and manufacturing. Screw connections of pipes loosened at two occasions which caused leaking of several liters of cooling fluid. A fan of the free cooler mounted on the roof of the MTF failed after

only 3 months of operation.The problems with the thermal subsystem caused temporary temperature and humidity issues inside the FEG.[16]

These problems can be minimized or fixed quickly by having a better monitoring system which enables the interaction and relation visualization of sensors and actuators data between different subsystems used in FEG. The current monitoring software infrastructure provided by Argus Controls which is currently in use do not provide such features. It would be problematic to use the ARGUS API directly by frontend visualization system due to latency and limited bandwidth available for EDEN ISS project. On the other hand, having a local database with all the sensor and actuator data solves the latency and bandwidth problem and open new possibilities for further data analysis steps.

My supervisor from EDEN ISS team provided me with the guidelines for the software that could solve the problems faced during last deployment seasons. They are listed below:

- Get sensor and actuator values from ARGUS API endpoints
- Store the received data in a database
- Periodically call the API endpoints and store updated data
- Design a web based GUI and visualize the interaction and relation of sensors and actuators data between different subsystems

In a nutshell, the API endpoints need to be called periodically to get data of 121 sensors and 99 actuators and store them in a database. Then the stored data need be served to a GUI interface for visualization purposes. The whole task can be further divided into two tasks, server side and client side interface. The server side software infrastructure is responsible to periodically call all the API endpoints and store the data in systematic order. Further more, it need to server the stored data which can be done by writing REST API on top of it. The client side interface is responsible to retrieve the data using the REST API of server side and visualize the data as per the layouts provided by EDEN ISS team. The server side system is called Backend system and the client side system is called Frontend system. This software project need a tool to handle client and server side web software infrastructure.

4 Tools and Technologies

To implement the software infrastructure which is capable of fulfilling EDEN ISS's requirements, the next step is to analyze the guidelines and decide on the programming language, tools, and technologies. There is no constrain to which programming language and tools need to be used to design and implement the project from the EDEN ISS team. While choosing the right tools, criteria like previous experience, suitability for the task, maintainability, performance, and so on were taken into consideration. After thorough analysis, the following programming languages, tools, packages, and libraries were considered for the project to implement the software according to the guidelines.

- Server Side System (Backend)
  - Python based Dango web framework [13]
  - TreeFrog Framework, High-speed C++ MVC Framework for Web Application[14]
  - Laravel, The PHP Framework for Web Artisans [12]

  - Gruzzle, A PHP HTTP client that makes it easy to send HTTP requests and trivial to integrate with web services[4]
  - Curl, a command line tool and library for transferring data with URLs[3]
- Client Side System
  - React, A JavaScript library for building user interfaces[10]
  - Angular, A JavaScript library for web web interfaces [8]
  - Jointjs, Opensource version of Rappid library used to create flowchart and other diagramming interfaces with ease[7]

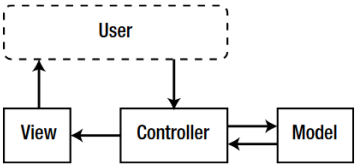
It is important to understand a tool in detail to get the most out of it. Having a grain detail knowledge of a tool and features offered by it, one can focus on business logic to tackle the use case rather than reinventing the wheel. Among the tools and libraries above, the Laravel web framework is chosen for the Server-side software infrastructure because of previous experience, detailed documentation, and suitability for the task compared to other web frameworks. Guzzle is used instead of cURL. Although both tools can get the job done, with Guzzle the same result can be obtained with few lines of code and provides superior features like asynchronous calls which might be relevant in the future. React Library is used to create client-side GUI interface. The goal is to integrate a JavaScript-based opens-source diagrams and graph visualization library called JointJs. It is observed that the DOM [a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document][6] management for React library is different than Jointjs. It is a challenge in itself to integrate these two libraries but the software requirement for layout visualization can be achieved by combining the features provided by these two libraries. A brief introduction to the selected tools and technologies is as below.

4.1 Laravel Web Framework

Laravel is a free and open-source web framework created by Taylor Otwell with the first beta version made available on June 9, 2011. The project is open-sourced under MIT License. Laravel is based on a model-view-controller architectural pattern. Laravel provides features to create accessible, yet powerful applications by providing powerful tools needed for large, robust web applications. It provides inversion of control container, expressive migration system, and tightly integrated unit testing support. The feature to create custom artisan commands in laravel comes handy for the project use case, enabling to integrate third-party resources with clean and minimal code. The model-view-controller (MVC) concept is used in the implementation section. A brief explanation of MVC is provided below taken from Chris Pitt's book called Pro PHP MVC.

MVC (Model-View-Controller) is a software design pattern build around the interconnection of three main components loosely termed as models, view and controller. The Model contains the core business logic of an application. Business logic can be anything specific to how an application stores data, or uses third-party services, to fulfill its business requirements. If the application need to store/access data in database, the code to do so would be in model. The view is where all of the user interface elements of our application are kept.

Figure 2: MVC in a nutshell



Anything a user sees or interacts with can be kept in a view. The controller connects models and views. Controllers isolate the business logic of a model from the user interface elements of a view, and handle how the application will respond to user interaction in the view.[9]

4.2 Guzzle

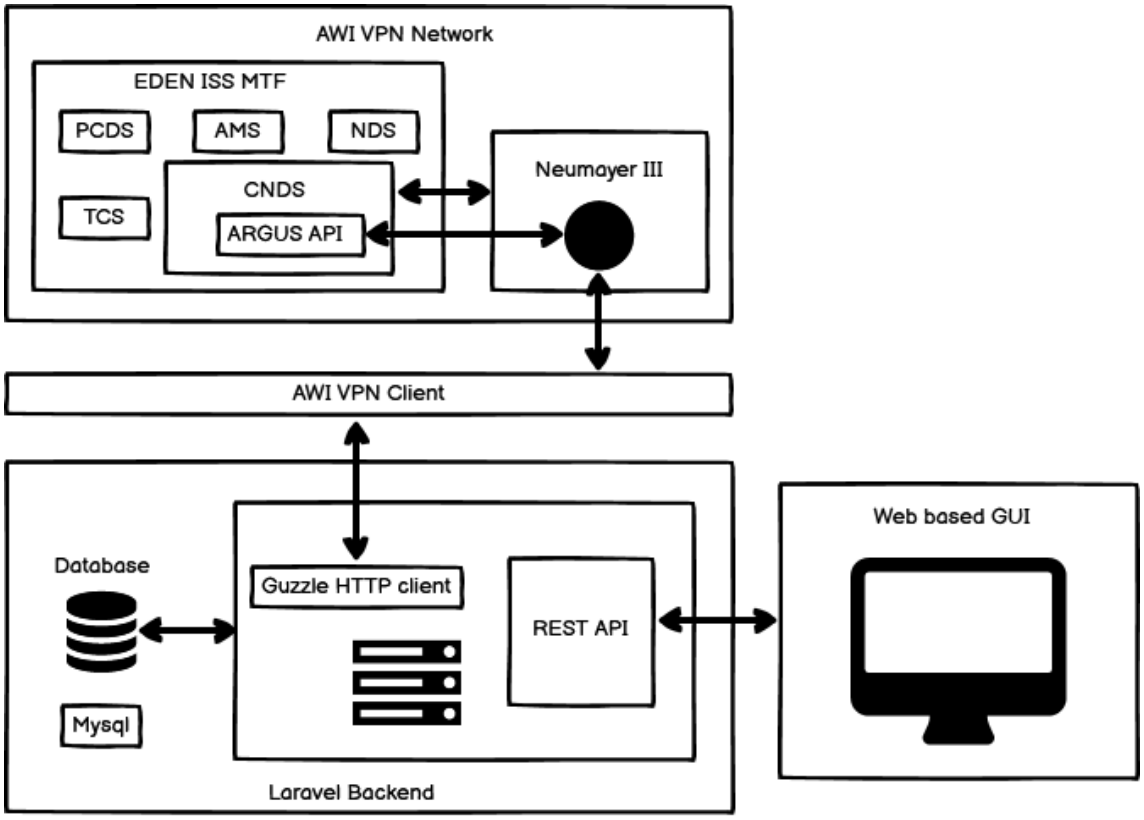
Guzzle is a PHP HTTP client library designed to send HTTP requests and can be easily integrated with laravel server side code. It can send both synchronous and asynchronous requests using same interface. It abstracts away the underlying HTTP transport, allowing to write environment and transport agnostic code i.e. no hard dependency on cURL, PHP steams, sockets or non-blocking event loops. The library is open-sourced under MIT License and has active support from the community.[4]

4.3 React

ReactJS is a JavaScript library used for building user interface. It encourages the creation of reusable UI components, which present data that changes over time. It is generally used as the V in MCV architecture. React abstracts away the DOM, offering a simpler programming model and better performance. React makes it painless to create interactive UIs. One can design simple views for each state in an application and react will efficiently update and render just the right components when the data changes.[10]

5 Design and Architecture

The guidelines and software requirements in section 3 gives brief overview on what need to be implemented as a code infrastructure. But, when there are multiple pieces involved in a software, it is overwhelming to not have a clear visual representation. A design and architecture phase helps to visualize the technical detail. It establish relation between different components of a software and how they interact with each other. It can be considered as a blueprint of the software. It gives basis to divide overall tasks into small tasks making it easy for the developer to tackle.



The software architecture of the project is as above. The EDEN ISS MTF is the source of sensor data. The CNDS connects the Neumayer III and MTF with wireless network. The ARGUS PC within CNDS serves the data with API infrastructure provided by ARGUS Systems. Both MTF and Neumayer III are protected and can be accessed only if connected with AWI VPN network. On the Laravel Backend side, I have Guzzle HTTP Client which calls the ARGUS API. The HTTP request goes through AWI VPN Client, and it gets data in HTTP response. The data is decoded and systematically saved in Mysql database. The REST API serves the stored data in the database. It is called from frontend system, a web based GUI in this case showing the real time sensor data. The Guzzle HTTP client is calling all the ARGUS sensor data API endpoints periodically and the data is being saved and served to Web based GUI. The software can show the latest sensor data points in the GUI, enabling better monitoring of EDEN ISS MTF.

6 Implementation

The implementation can be divided into two systems, server side (Backend system) and client side (Frontend system). This section explain the implementation steps for both system considering the project progress obtained in the end.

6.1 Understanding ARGUS API

The source of sensor data for the backend system is ARGUS API. It is the interface to ARGUS System database of data points which can be used to retrieve data. It uses REST API structure, a Representational State Transfer API that uses HTTP requests to GET, PUT, POST and DELETE data. ARGUS API currently serves GET requests only which allows to receive data (read only) in JSON string format. [2]

When handling large amount of data, ARGUS API return multiple messages to a single GET request. It uses a single thread to process the requests. Each connection has its own thread and when the GET request is received, this thread verifies the data (IDs, time frame) and adds them to a queue of elements to get processed. The reception threads may also need to split the requests by time to ensure that nothing takes too long. This ensures the regular access is not starved by this request. The process is multi threaded - when a request is received it gets threaded onto a worker thread and the listening thread goes back to listening. [2]

The ARGUS API have a authentication layer and HTTP client need to authenticate with valid Argus **Username** and **Password** for the ARGUS server to return data for respective HTTP request. The configuration and credentials for ARGUS API can be managed from ARGUS Operator software. A listener thread spawns a proxy thread when connected to the ARGUS server. When a request is received, the proxy completes the action. Each connection has it's own proxy and should not be waiting for the other connections. One thread processes all access to the data stream - this is the way that it is handled with the current graph. [2]

The ARGUS API has a **Parameter ID** for each sensor, a unique identity of each sensor in the+ ARGUS System. The API has an endpoint that can return all the available sensors, actuators, and other items whose data can be retrieved using the API. When HTTP request is sent, it returns a list of groups and items being data-streamed that can be used to parse parameter id for the sensors of choice. However, due to the complexity of parsing the returned data, this project is not making use of this feature. [2]

The API's URIs (Uniform Resource Identifiers) has the following structure: The IP Address or Computer Name / API Name - always ARGUS API / Version # / Query (optional)

For example: **ArgusServer:47840/argusapi/V1/parameters/1234**

The above example have **ArgusServer** as Computer name. The IP address of the computer can be used as well. The Port number used to access the computer is **47840**. **argusapi** is the name of the API. The version of the API is **V1**. The section after version is a query. In this case, **parameters** indicates that the API want to access the parameter data. The number followed by parameters is a unique **Parameter Id** for a sensor or an actuator, which is will return the latest data stored in the server.

There are additional queries modes available allowing to get data between time frame using date, for last few hours and the current values. The example for different type of queries possible is listed below. [2]

- **ArgusServer:47840/V1/parameters/7654?startDate=yyyy/mm/dd-hh:mm:ss&endDate=yyyy/mm/ss-hh:mm:ss**
- **ArgusServer:47840/V1/parameters/9843?startDate=yyyy/mm/dd-hh:mm:ss&endDate=yyyy/mm/ss-hh:mm:ss&metric**
- **ArgusServer:47840/V1/parameters/9843?startDate=yyyy/mm/dd-hh:mm:ss&endDate=yyyy/mm/ss-hh:mm:ss&imperial**
- **ArgusServer:47840/ArgusApi/V1/parameters/9843?from=2h&to=1h**
- **ArgusServer:47840/ArgusApi/V1/parameters/9843?from=2h**
- **ArgusServer:47840/ArgusApi/V1/parameters/9843?from=2h&limit=100**

- **ArgusServer:47840/ArgusApi/V1/parameters/9843?currentValues**

  The last version mentioned above is used in this project. The idea is to synchronize the data in server and the software's database. In addition, there is a possibility to send multiple **Parameter Id** in single request which will reduce the number of request made over time. This feature is not used in current implementation. This can be a good way to optimize the performance of the software in future.

A sample ARGUS API call look may look as following.

URI: ArgusServer:47840/ArgusApi/V1/parameters/40174?currentValues


```
{
    "DataList": [
        {
            "Parameter": 40174,
            "DataSet": [
                {
                    "Data": "0",
                    "Time": "Tue Sep 27 13:58:46 2016"
                }
            ]
        }
    ]
}
```


[15]

6.2 Backend System

The backend system handle retrieval, storage and serving steps of the data extraction pipeline. The details on various components of backend system and how they interact with each other to extract, store and serve the sensor data to frontend system in explained in this section.

6.2.1 Database Migration

Database migration is a feature provided by the Laravel framework which makes it easier to create and manage database schema. Migrations are typically paired with Laravel's schema builder to build your application's database schema. The laravel Schema facade provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems. MySql database is used for this software project. MySql is an open-source relational database management system. A relational database manages and organizes the data in one or more tables with the possibility to create a relationship between data type and tables - a concept that helps to structure the saved data.

A generalized blueprint for the sensors and actuators types is created making it easier to create tables and systematically structure the database. A total of 17 categories for sensors and actuators is created. A table below shows the total number of categories with their name and number of sensors enlisted in that category.

S.N	Category	No. of Sensors	Data table
1	EC Sensors	6	EC Data
2	Fans	5	Fan Data
3	Flow Meters	2	Flow Meter Data
4	Gas Sensors	4	Gas Data
5	Pumps	8	Pump Data
6	HP Pump values	16	HP Data
7	Intensity Controls	4	Intensity Data
8	Leak Sensor	2	Leak Data
9	Level Switches	5	Level Switch Data
10	Ph Sensors	6	Ph Data
11	Pressure Sensors	11	Pressure Data
12	Rh Sensors	5	Rh Data
13	Sensor Switch	3	Sensor Switch Data
14	Solenoid	5	Solenoid Data
15	Temperature Sensors	22	Temperature Data
16	Valves	3	Valves Data
17	Level Sensors	6	Level Data
Total	17	113	17

Each category has a sensor table that stores all the sensors that fall under that category with their **label** and **parameter Id**. The data for each of these sensors are stored separately in the sensor data table. The sensor table and sensor data table are in one-to-many relationships. To create this relationship, the sensor id from the sensor table is used in the sensor data table as a foreign key. An example of a table schema for the sensor and sensor data table is as follows.

```
// Gas Sensor Table
public function up()
  {
      Schema::create('gas_sensors', function (Blueprint $table) {
          $table->bigIncrements('id');
          $table->string('parameter_id');
          $table->string('parameter_label');
          $table->timestamps();
      });
  }
```

Through out the project, all the sensor table have same structure. There are four columns, a unique id, parameter id for ARGUS API, name of parameter as parameter label and timestamps.

```
\\ Gas Sensor Data Table
public function up()
  {
      Schema::create('gas_data', function (Blueprint $table) {
          $table->id();
          $table->unsignedBigInteger('gas_sensor_id');
          $table->string('parameter_id');
          $table->string('data'); // $table->string('state');
          $table->string('recorded_time');
```

```
            $table->foreign('gas_sensor_id')
                ->references('id')
                ->on('gas_sensors')
                ->onDelete('cascade');
            $table->timestamps();
        });
    }
```

The schema for the sensor data table is very much similar for all the sensor data tables mentioned above in a table with an exception to the data column. A state column is used instead in those cases as shown above.

A laravel seeder is used to seed the sensors belonging to the sensor table. This is because while calling the ARGUS API, the sensor parameter id is retrieved from a respected sensor table. Although ARGUS API provides an endpoint that can return parameter ids, the JSON data received is complicated to parse. Depending on the API to get parameter ids would result in not being able to call other sensor data endpoints if any problem arises with this endpoint or problem with parsing creating a single point of failure. An example of a sensor seeder class is as follows.

```
\\Gas Sensor Seeder
class GasSensorSeeder extends Seeder
{
    const PARAMETER_IDS = ['Co2 Omni box 1' => 11728,
    'Co2 Omni box 2' => 11732, 'co2 targets' => 11803,
    'o2' => 1403];
    public function run()
    {
        GasSensor::all()->isEmpty() ? $this->seedMe(): true;
    }

    private function seedMe()
    {
        foreach (self::PARAMETER_IDS as $idKey => $idValue)
        {
            $gasSensor = new GasSensor();
            $gasSensor->parameter_id = $idValue;
            $gasSensor->parameter_label = $idKey;
            $gasSensor->save();
        }
    }
}
```

6.2.2 Eloquent ORM (Model) and relationship

Object-relation-mapping (ORM) is the idea of being able to write SQL queries and interacting with a relational database using the object-oriented paradigm of your preferred programming language instead of SQL. It abstracts away the database system so that switching from a different relational database is easier. ORM also provides advanced features like database transactions, connection pooling, migrations, seeds, streams, and

so on out of the box. [5] The laravel Eloquent ORM is an ORM provided by the Laravel framework. Each database table has a corresponding ”Model”, a part of MVC architecture as introduced before, which is used to interact with that table. Models allow to query for data in the tables as well as insert new records into the table.[12]

Database tables are related to each other. After a relationship is established between tables in the migration file as presented section 6.2.1, eloquent relationships are defined as methods on Eloquent model classes. Eloquent relationships are defined as methods in a model. Like Eloquent models themselves, this serve as powerful query builders, method chaining and querying capabilities.[12]

With Eloquent ORM, a one-to-many relationship is established between the sensor model and sensor data model by adding *hasMany* method in sensor Model and *belongTo* method in the sensor data model. An example of a gas sensor and a gas sensor data relationship is as follows.

```
\\GasSensor Model class
 class GasSensor extends Model
 {
    public function data()
      {
          return $this->hasMany(GasData::class);
      }
 }

 \\GasData Model class
 class GasData extends Model
 {
     public function sensor()
     {
         return $this->belongsTo(GasSensor::class);
     }
 }
```

6.2.3 API Call service

The procedure to call the ARGUS API is a core business logic of the software and resides on respective Sensor Data Models. There are altogether 17 data models that need to call more than 113 ARGUS API endpoints. The generalization of sensor Parameter Ids into different categories was an impactful decision, which solves redundant code blocks problem while calling the API endpoints. However, the APIs need to be called from 17 data models resulting in code duplication for API caller. To avoid this situation, a service class is written which is a reusable block of code that can be shared among all the 17 models instead of including the same code inside them. This reusable block of code which is only responsible to call the ARGUS API and return the received data is API Call Service. The API call service can be instantiated in data models and can be used to call ARGUS API. The API call service is as follows.

```
\\API call service
use GuzzleHttp\Client;
```

```
class ArgusApiCallService
{
    public function CallApi($parameter_id)
    {
        $client = new Client(['verify' => false ]);
        try{
            $response = $client->request('GET', env('ARGUS_URL') .
            $parameter_id . '?currentValues',
            ['auth' => [env('ARGUS_USERNAME'), env('ARGUS_PASSWORD')]]);
        } catch (Exception $exception)
        {
            echo $exception;
            return $exception;
        }
        $body = $response->getBody();
        $decoded_body = json_decode($body);
        if (count($decoded_body->DataList) == 0){
            return $decoded_body;
        }
        return $decoded_body;
    }
}
```

In the API call service, the Guzzle library is used to make an HTTP GET request. In the beginning, a client has been instantiated and is used to send GET requests. While sending a GET request, the URI for the request is passed as a parameter to the request method and API authentication username and password are passed inside an array with a key-value mapping of *'auth'* keyword and an array of username and password. The final URI in this case look like this: **ArgusServer:47840/argusapi/V1/parameters/**{**parameterId**} where **parameterId** is the function parameter. The try-catch block is used to catch possible exceptions. Out of request-response, the body is extracted and JSON is decoded using the PHP json_decode() function. The ARGUS API gives empty JSON in case there is no data. Those cases are handled by checking if the data is empty and only returned if it is not.

The Username and Password for ARGUS API and ARGUS URL are saved in the environment file and retrieved from there during run time.

6.2.4 JSON Parser Service

The JSON parser service is called in a data model right after the API call service. The data returned by API call service may look like the JSON data attached below.. The main goal of the JSON parser service is to select the relevant data from JSON and store it in a temporary data structure which is a Laravel collection in this case.

Laravel Collection is a data structure that provides a fluent, convenient wrapper for working with arrays of data. The collection allows chaining its methods to perform fluent mapping and reducing of the underlying array. They are immutable, meaning every Collection method returns an entirely new Collection instance.[12]

The data stored in the temporary collection is returned from the Decoder method in JSON

```
{
    "DataList": [
        {
            "Parameter": 40174,
            "DataSet": [
                {
                    "Data": "0",
                    "Time": "Tue Sep 27 13:58:46 2016"
                }
            ]
        }
    ]
}
```


Figure 3: Returned data from API Call service

Parser service and used in the sensor data model to store the data points in the database. As can be seen in the JSON Object attached above, the object inside ”DataSet” has either ”Data” or ”State” in different cases. Due to this, there are two JSON parser service i.e. Dataable Service and Stateable Service. They are responsible to extract ”Data” and ”State” respectively. The decision to create these services resulted in a clean and maintainable code and reduce the duplication of code in all the data models. The JSON parser service (Dataable Service) looks as follows.

```
class DataableService
{
    public function DataableDecoder($data)
    {
        try{
            $end_data = collect();
            $collect_data = collect($data)->collapse();

            $parameter = $collect_data->pluck('Parameter');
            $end_data->put('parameter_id', $parameter);

            $data_set = collect($collect_data->pluck('DataSet'))
            ->collapse();
            $sensor_data = $data_set->pluck('Data');
            $recorded_time = $data_set->pluck('Time');

            $end_data->put('data', $sensor_data);
            $end_data->put('recorded_time', $recorded_time);
            return $end_data;
        }
        catch (Exception $exception)
        {
            report($exception);
```

```
            return false;
        }
    }
}
```

In the service class, the decoder function takes data that need to be decoded as a function parameter. The main code block is inside the try block to catch possible exceptions. A temporary laravel collection is instantiated in the beginning. The *collapse method* of collection collapses a collection of arrays into a single, flat collection. The collapse method is called twice in each Decoder method. When the collapse method is called the first time, it brings ”DataList”, ”Parameter” and ”DataSet” in the same object collapsing the array in between. The ”Parameter” value is extracted using *pluck method* and stored in the temporary collection as a key value. Once the parameter id is extracted, the *collapse method* is called the second time to collapse ”DataSet”. Following this step, ”Data” value or ”State” value in the case of Dataable service or Stateable service respectively along with ”Time” are extracted using *pluck method* and saved to the temporary collection as key-value.

The temporary collection may look like this when it is returned from the decoder function.

```
\\Dataable Service
['parameter_id' => '1234', 'data' => '10%',
'recorded_time' => 'Tue Aug 02 12:32:14 2020']

\\Stateable Service
['parameter_id' => '1234', 'state' => '0',
'recorded_time' => 'Tue Aug 02 12:32:14 2020']
```

6.2.5 Storing Parsed Data

Storing parsed data is a final step of the data extraction pipeline. The API call service is explained in detail in section 6.2.3 but this is where that service is used. The API call service takes parameter id as a function parameter. Another intermediary private function called *getProcessedData* belonging to the data model class is used in every data model. This function takes a parameter id as a function parameter. An example of *getProcessedGasData* is as following.

```
private function getProcessedGasData($parameter_id)
  {
      $api_service = new ArgusApiCallService();
      $dataable_service = new DataableService();
      $gas_data = $api_service->CallApi($parameter_id);
      if (count($gas_data->DataList) == 0)
      {
          var_dump('The parameter id: ' . $parameter_id .
          ' gives no data. ');
          return null;
      }
      else
      {
          $decoded_data = $dataable_service
```

```
            ->DataableDecoder($gas_data);
            return $decoded_data;
        }
    }
```

The **getProcessedGasData** function is written separately to simplify the logic where each function is doing one and only one task resulting in clean and maintainable codebase. In first few lines, API call service and Dataable JSON parser service is instantiated. Then the API service is called and JSON data is received. If the data is empty a log is printed that the received data is empty. The received data from API call service is parsed using Dataable JSON parser service and result is returned.

In each data model, there is a function called *newData* which is used to store the received data. An example of *newGasData* function is as following.

```
public function newGasData()
  {
      $nested_sensors = (GasSensor::all('id', 'parameter_id'))
      ->toArray();
      $sensor_parameter_ids = Arr::pluck($nested_sensors,
      'parameter_id');
      $sensor_ids = Arr::pluck($nested_sensors, 'id');

      $sensors = array_combine($sensor_ids, $sensor_parameter_ids);
      foreach ($sensors as $sensor_id => $parameter_id)
      {
          $decoded_data = $this->getProcessedGasData($parameter_id);
          if ($decoded_data !== null)
          {
              try{
                  $gas_data = new GasData();
                  $gas_data->gas_sensor_id = $sensor_id;
                  $gas_data->parameter_id = $decoded_data
                  ->get('parameter_id')->first();
                  $gas_data->data = $decoded_data->get('data')
                  ->first();
                  $gas_data->recorded_time = $decoded_data
                  ->get('recorded_time')->first();

                  $gas_data->save();
                  var_dump( 'saved successfully !! ' . 'Parameter Id: '
                  . $decoded_data->get('parameter_id')->first());
              }
              catch (Exception $exception)
              {
                  report($exception);
                  return $exception->getMessage();
              }
          }
      }
      return true;
```

```
    }
```

The decoded data is returned from the decoder function of the respective JSON parser service used in the data model. This returned data is a key-value collection array as shown at the end of the previous section. The sensor parameter id and sensor id is retrieved from the sensor table in the database and stored in a temporary key-value array. A for loop is used to loop the sensor parameter id and sensor id in a key-value array. The decoded data is received by calling *getProcessedData* function which have parameter id as function parameter. An if-block is used to check if the decoded data is null and if not core storing step is performed inside try-catch. An exception is thrown from the try-catch block if any error occurs while writing to the database. In the try block, a new data model is instantiated and the next steps are pointing a column in the database and storing a data point. The id of the sensor table is a foreign key in sensor data, hence it is being pointed in gas_sensor_id column in this case. Similarly, the value of parameter id is retrieved from decoded data and pointed in parameter_id column, data in the data column, and recorded time in recorded_time column. Finally, the save method is called to store the pointed data in respective database columns. The *newData* function is a public function and is used in corn job class to do the data extraction periodically.

6.2.6 Periodic APIs call

Laravel provides a feature to create custom commands and corn job scheduler to execute the available commands periodically. The customs commands are in `app/Console/Commands` directory of the codebase. A command for each data model has been created in addition to a command called *allApiCalls* which has all the sensor data commands. The *newData* function can be called from a custom data model command which will call the ARGUS API and save the returned data for all the sensors in the sensor table. An example of a gas data model command is as follows.

```
class gasDataApiCall extends Command
{
    protected $signature = 'argus:gas-data'; //command name
    and signature

    public function __construct() //create new command instance
    {
        parent::__construct();
    }

    public function handle() //execute console command
    {
        $this->info('Calling all ARGUS API endpoints for Gas Data');
        $gas_data = new GasData();
        $gas_data->newGasData();
        return true;
    }
}
```

The command name is used to call the command from terminal or execute command. The core logic resides inside *handle* function. In *handle* function, a new instance of

sensor data model is created. Then *newData* function for respective sensor data model is called. The *allApiCalls* command have other sensor data commands inside its *handle* function. A sensor data command is called inside *handle* function of *allApiCalls* as `Artisan::call('argus:gas-data');`.

The next step is to schedule the periodic calling of *allApiCalls* command. The scheduler function provided by laravel resides in `Kernal.php` file in `app/Console` directory of the codebase. An example of *schedule* function is as following.

```
protected function schedule(Schedule $schedule)
  {
        $schedule->command('call:argus')->everyThirtyMinutes();
        //$schedule->command('call:argus')->everyFifteenMinutes();
        //$schedule->command('call:argus')->everyFiveMinutes();
        //$schedule->command('call:argus')->everyMinute();
  }
```

The *schedule* function takes an instance of `Illuminate\Console\Scheduling\Schedule` as a function parameter. The `'call:argus'` is the name of *allApiCalls* command. The first line of *schedule* function is calling *allApiCalls* command every thirty minutes. Alternative calling periods are listed as commented code just below which includes every fifteen minutes, every five minutes and every minute.

The schedule and sensor data commands are designed as modular components. The software can be used, the result can be analyzed, and calling frequency can be updated according to the requirement. There is no need to change code in any other parts of the software. This made the software easy to use and maintain over time.

Instead of calling *allApiCalls* command, the individual sensor data commands can be called in *schedule* function with the possibility to have different command call interval. This point is listed below in the future work section.

6.2.7 RESTful API Infrastructure

The software guidelines from the EDEN ISS team include the design and implementation of a web-based GUI which helps to visualize the interaction and relation of sensor data between different subsystems. After completion of section 6.2.6, the data is successfully extracted and stored systematically in the database. The frontend system or the V of MVC architecture used in the software needs a way to access data stored in the database. An interface needs to be implemented which let the frontend system to easily access the data in the database and visualize as per the layout designs provided by the EDEN ISS team.

The interface to serve the sensor data to the frontend system is RESTful API infrastructure for each sensor data model. Three different core components are used to implement sensor data RESTful APIs with clean and maintainable codebase. They are explained below with their functionality and working mechanism.

JSON Resource:

The API resource is a laravel feature define as a transformation layer that sits between Eloquent models and the JSON responses of the API. Laravel resource classes allow

to expressively and easily transform model and model collections into JSON data[12]. API resource is used in the controller to convert model collection data retrieved from the sensor data table into a structured JSON data that the API endpoint will be returned. A separate laravel resource is implemented for each sensor data model. An example of laravel resourced used in the software is the following.

```
  class GasDataResource extends JsonResource
{
    public function toArray($request)
    {
        return [
            'id' => (int) $this->id,
            'parameter_id' => (string)$this->parameter_id,
            'data' => (string)$this->data,
            'recorded_time' => (string)$this->recorded_time,
        ];
    }
}
```

The JsonResource is an instance of `Illuminate\Http\Resources\Json\JsonResource` class. In case gas data, four parameters are included i.e. id, parameter_id, data, recorded_time. A 'state' will be returned instead of data if the data model have state instead of 'data'.

Controller:

The controller is the C component of MVC architecture and connects the sensor data model with the frontend system which acts as a view component of MVC. There is a controller for each sensor model and sensor data model. With the current version of the software, only sensor data models are being used by the frontend system. An example of a data controller class is as follows.

```
class GasDataController extends Controller
{
    public function index()
    {
        return GasDataResource::collection(GasData::all());
    }

    public function dataWithParameterId($parameterId)
    {
        return GasDataResource::
        collection(GasData::gasDataWithParameterId($parameterId));
    }
}
```

There are two functions in each sensor data controller i.e. *index* function and *dataWithParameterId*. The index function is used to get all data from the sensor data model and return it. As in the above example, all gas data is queried and wrapped with gas resource collection. This step is retrieving all the gas data saved in the database and returned as per the JSON structure defined in the data resource. Pagination can be used to limit the number of data returned by the index function if needed. The next function *dataWithParameterId* takes parameter id as function parameter and return the latest recorded

sensor data. The query to do so is written as a scope -a feature in laravel, in each data model. This is the data that needs to be constantly updated in the frontend visualization layout. An example of scope is as follows.

```
public function scopeGasDataWithParameterId($query, $parameterId)
{
    return array($query->whereIn('parameter_id',
    [$parameterId])->orderByDesc('recorded_time')->first());
}
```

Because a sensor data table stores data for multiple sensors in the same category having different parameter Id, the query is looking for the passed parameter id and ordering the recorded time of sensor data in descending order and getting the first entity. Both functions in the controller will be directed to a specific route which is explained in the route section below.

API Routes:

Routes are defined URI in the web application which when visited, a code block defined in that route or function called inside the route will be executed. Each unique route is used to call a function in the controller in case of this software. A function parameter required for a function in the controller can be passed using routes. In this software project, each function in each of the controller has a unique route defined in `routes\api.php` file. An example of a route is as follows.

```
Route::get('gas-data', 'GasDataController@index')
->name('gas-data.index');
Route::get('gas-data/{parameterId}',
'GasDataController@dataWithParameterId')
->name('gas-data.show');
```

Both of the routes above are `GET` routes. Laravel have six route options available i.e. get, post, put, patch, delete, and options. If the route is defined as a GET route, the HTTP client needs to send a `GET` request to receive the response and so on for all the other route types. The first route above is the calling *index* function in Gas Data Controller. While the second route is calling *dataWithParameterId* function in Gas Data Controller with the function parameter. For each sensor and sensor data controllers, there is a unique route defined. With the routes defined, the RESTful API infrastructure is complete and ready to serve data to the respective HTTP client request when called with proper syntax.

Detailed documentation of all the RESTful APIs is generated automatically using a laravel package called Scribe[11]. The documentation includes details on how to use the APIs and examples with cURL and JavaScript.

6.3 Frontend System

The ReactJs frontend system is the core of the user interface of the software project. There are two core tasks in the frontend system i.e. retrieving the necessary data by calling respective RESTful API and visualizing the interation and relation of sensor data and different subsystems. A ReactJs dashboard starter from Creative Tim called Argon Dashboard PRO is used in the project.[1] The code in frontend system is added on top of the starter template to reduce the design complexity and focus on key business logic

to achieve the project goal. The two sections of frontend systems are explained in brief below.

6.3.1 Calling RESTful API

A react component called `useApiFetch` is implemented to fetch the data from the RESTful APIs. The `useApiFetch` is a functional react component which takes the API URI and use JavaScript *fetch* function asynchronously to get the JSON result. A `setData` state function is used to put received JSON in the 'data' state which is returned from the component along with possible errors caught from the try-catch block. A separate functional component is implemented for each sensor data category used in the system which takes parameter Id as a function parameter. The parameter Id is fused with the API URL to call the respective sensor data route and the response object is saved in a variable. The necessary information can be parsed from the variable and used in an appropriate place in the visualization layout. Hence, a loop to retrieve data from the Backend system using RESTful APIs is completed.

6.3.2 Layout Visualization

As per the EDEN ISS team's guidelines and requirements, the visualization part is not completed in the meantime. There is three visualization layout for the different subsystem. An example of a Thermal Control System layout is as follows.

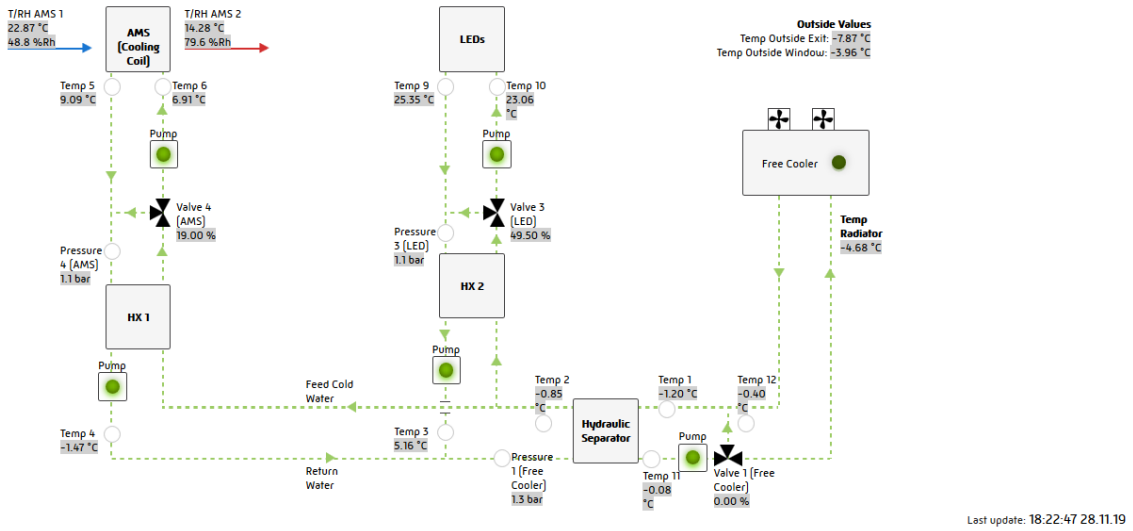


Figure 4: Block Diagram Thermal System

As per guidelines (section 3), a web-based GUI to visualize the above layout along with the other two layouts i.e. Air management System layout and Nutrient Delivery System was expected. The main problem faced while implementing a web-based GUI visualization layout is combining the JointJs library and ReactJs. As mentioned in the 4 section, the data DOM in JointJS and ReactJS are different however there is a possibility to combine them. The JointJs library uses the DOM element to dynamically access and updates the content. The ReactJs uses props to pass data from parent components to children components. When using JointJS and ReactJS, the data is being modified imperatively

outside of the typical dataflow. For these kinds of scenarios, React Refs, a react feature that provides a way to access DOM nodes or React elements created in the render method, can be used. It is not a straight forward data flow but a hack to integrate React with third-party DOM libraries. ReactJS have detail documentation on how to work with Refs but there is an unclear resource from JointJs library. After thorough implementation attempts and spending significant time to go through the documentation of both libraries, the expected result was not achieved in this case.

7 Testing

Laravel provides a PHPUnit testing environment out of the box. The `phpunit.xml` with all the testing environment configuration is set up while creating laravel application in the beginning. The framework also ships with convenient helper methods that allow to expressively test laravel application. All the laravel tests reside in the tests directory. Two types of tests can be implemented for an application i.e. Feature and Unit. Unit tests focus on a very small, isolated portion of code or even a single function. Feature tests, however, test a large portion of code, including how several objects interact with each other or even a full HTTP request to a JSON endpoint.[12]

The feature tests have been implemented primarily to test the sensor data RESTful APIs. Laravel provides a very fluent API for making HTTP requests and examining the output which is used in the project while implementing feature tests. An example of a sensor data feature test class is as follows.

```
class GasDataControllerTest extends TestCase
{
    use RefreshDatabase;
    private $gas_data;

    protected function setUp(): void
    {
        parent::setUp();
        $this->gas_data = factory( GasData::class)->create();
    }

    public function testIndex()
    {
        $response = $this->get(route('gas-data.index'));
        $response->assertStatus(200);
    }

    public function testShow()
    {
        $response = $this->get(route('gas-data.show',
        $this->gas_data->parameter_id));
        $response->assertStatus(200);
        $response->assertJsonFragment(['id' =>
        $this->gas_data->id]);
        $response->assertJsonFragment(['parameter_id' =>
        $this->gas_data->parameter_id]);
```

```
            $response->assertJsonFragment(['data' =>
            $this->gas_data->data]);
            $response->assertJsonFragment(['recorded_time' =>
            $this->gas_data->recorded_time]);
        }
}
```

For each API route, a function in the sensor data controller is defined which returns a response when the route URI is called. It is vital to test the response status by calling each API. As in the example above, Each test uses the Refresh Database trait class which restores the database to the prior state after the test is run. A dummy gas data is created in *setUp* function and stored to private `gas_data` variable. The two later functions tests *index* and *show or dataWithParameterId* function in gas data controller. The relevant get routes are called and the status of the response is asserted as `200` HTTP code which means the request has succeeded. Also, the returned JSON fragment can be tested. In this case, it should be identical to the data in `gas_data` variable. A similar testing format is used to test all the sensor data RESTful APIs.

The Unit tests for the ARGUS API call service may be required to test the stability of the feature. The scale of the software and implementation complexity for two separate systems made it very difficult to implement all the test cases which can make the software robust. More details on testing suggestions are listed on 9 section below.

8 Conclusions

The data extraction and visualization system aims to improve the remote monitoring and subsystem problem debugging capabilities of the EDEN ISS project. The software project has been designed and implemented by following the use-case and guidelines provided by the EDEN ISS team. The data extraction pipeline is implemented using the PHP laravel framework which is capable of periodically calling ARGUS API to extract data from EDEN ISS MTF in Antarctica, store the data systematically in the database locally in DLR Bremen, and serve the stored data using modern RESTful APIs. The data extraction backend system is designed and implemented with a clean and modular architecture in mind. The time-period to call the ARGUS API can be modified easily. The different time-period can be set while calling ARGUS API for different sensor data.

The Backend system RESTful APIs is called successfully using the React frontend system completing the data flow loop for the extraction pipeline. However, few problems were faced while implementing the subsystem layout visualization as explained in section 6.3. Thorough research was carried to gather details about the faced problems and necessary further steps were identified to possibly solve the problems. The proposed solutions faced while implementing the subsystem layout visualization are listed in section 9 below.

9 Future Work

There are numerous ways to improve the current implementation of the software project. With utmost priority is the task to implement the visualization layout and display the fetched data in the frontend system in an appropriate place. A problem was faced as explained in section 6.3.2. Although a possible solution was identified but has not been

implemented due to time constraints. In case the approach used in this project (6.3.2) is complicated to start over with, an alternative approach is to not use the React library at all and implement the layout visualization using normal javascript and JointJs library. A small implementation is tested to combine JointJs with normal javascript after unsuccessful attempts with ReactJS and they worked well together.

In the frontend system, while calling RESTful APIs, a timer function needs to be implemented to periodically refresh the data received on the frontend side. As the core requirement to visualize layouts is not completed, this functionality is not implemented yet.

There are many ways to improve the speed and performance of the backend system as well. As mentioned at the end of section 6.1, there is a possibility to send multiple **Parameter Id** in a single request, while calling ARGUS API, which will reduce the number of requests made over time. Minor changes in the architecture of the data storing pipeline would be needed but this may improve the performance of the software.

A feature to log all the API calls both successful and unsuccessful with the time taken in log files may be implemented to have a better overview of the performance of the backend system. If this feature is extended to parse the saved log files with a user interface that can generate a performance report for each month, that will help to understand the performance of the system and optimize it over time.

References

[1] *Argon Dashboard PRO by Creative Tim*. URL: https://www.creative-tim.com/product/argon-dashboard-pro.

[2] *Argus API Programmer's Guide*. Jan. 2019. URL: http://www.arguscontrols.com/.

[3] *Everything curl*. URL: https://curl.haxx.se/.

[4] *Guzzle, php http client*. URL: http://docs.guzzlephp.org/en/stable/index.html.

[5] Mario Hoyos. *What is an ORM and Why You Should Use it*. Mar. 2019. URL: https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a.

[6] *JavaScript HTML DOM*. URL: https://www.w3schools.com/js/js_htmldom.asp.

[7] *JointJs: Visualize and interact with diagrams and graphs*. URL: https://www.jointjs.com/.

[8] *One framework. Mobile desktop.* URL: https://angular.io/.

[9] Chris Pitt. *Pro PHP MVC*. URL: http://www.web-algarve.com/books/MySQL%20&%20PHP/Pro%20PHP%20MVC.pdf.

[10] *React – A JavaScript library for building user interfaces*. URL: https://reactjs.org/.

[11] *scribe documentation*. URL: https://scribe.readthedocs.io/en/latest/.

[12] *The PHP Framework for Web Artisans*. URL: https://laravel.com/.

[13] *The web framework for perfectionists with deadlines*. URL: https://www.djangoproject.com/.

[14] *TreeFrog Framework*. URL: https://www.treefrogframework.org/.

[15] Vincent Vrakking et al. *Service Section Design of the EDEN ISS Project*. July 2017.

[16] Paul Zabel and Conrad Zeidler. *EDEN ISS: A Plant Cultivation Technology for Spaceflight.* May 2019. DOI: 10.1007/978-3-319-09575-2_210-1.

[17] Paul Zabel et al. *Future Exploration Greenhouse Design of the EDEN ISS Project*. July 2017.